IT Knowledge Bases
itknowledgebases.com

CVE-2026-2441

Google Chrome CSS Use-After-Free Zero-Day

Security Vulnerability Analysis & Remediation Guide

Published: February 2026 | Severity: HIGH (CVSS 8.8) | Status: Actively Exploited

⚠ **CRITICAL**

This vulnerability is under active exploitation in the wild. CISA has added it to the Known Exploited Vulnerabilities (KEV) catalog with a mandatory patch deadline of March 10, 2026. Patch immediately.

1. Vulnerability Summary

CVE ID	CVE-2026-2441
CVSS Score	**8.8 (HIGH)**
Vulnerability Type	Use-After-Free (CWE-416) in CSS Engine
Affected Component	Google Chrome CSS Processing Component
Affected Versions	Chrome prior to 145.0.7632.75 (Windows/macOS) and 144.0.7559.75 (Linux)
Patched Versions	145.0.7632.75 / 76 (Win/Mac) | 144.0.7559.75 (Linux) | Extended Stable: 144.0.7559.177
Discovered By	Shaheen Fazim (reported February 11, 2026)
Patch Released	February 13, 2026
Exploitation Status	**Actively exploited in the wild (zero-day)**
CISA KEV	Added February 17, 2026 — Federal patch deadline: March 10, 2026
Also Affects	Microsoft Edge, Brave, Opera, Vivaldi (all Chromium-based browsers)

2. The Why — Root Cause Explained

2.1 What Is a Use-After-Free Vulnerability?

A Use-After-Free (UAF) is a class of memory corruption vulnerability rooted in how C++ manages heap memory. The pattern works like this:

1. The program allocates a block of memory for an object — in this case, a CSS font feature map.
2. The program frees (deallocates) that memory, signaling it is no longer needed.
3. A dangling pointer — a reference that still points at the now-freed memory address — remains in use.
4. When the program tries to access data through that dangling pointer, it reads or writes to memory that has since been reallocated for a different purpose.
5. An attacker crafts input that controls what gets placed in that recycled memory, effectively hijacking the program's logic.

In mature C++ codebases like Chromium's rendering engine (Blink), this class of bug is particularly common because the engine manages enormous numbers of short-lived objects — style rules, layout nodes, and rendering artifacts — all of which are created and destroyed at high frequency during page rendering.

2.2 Where Exactly Did the Bug Live?

The vulnerability was traced to Chrome's CSS font feature processing code — specifically in the function responsible for iterating over a FontFeatureMap (a data structure that holds custom CSS font property rules such as font-feature-settings or @property declarations).

The vulnerable code pattern looked roughly like this:

```
void ProcessFontFeatures(FontFeatureMap& map) {
    for (auto& rule : map) {      // Iterating over 'map' directly
        ApplyStyle(rule);          // This can TRIGGER a layout event
        // ApplyStyle() can cause 'map' to be freed mid-loop!
    }                              // <-- Dangling pointer dereference here
}
```

The problem: ApplyStyle() internally triggers a cascade of style recalculation events. One of those recalculations can cause the CSS property map object itself to be garbage collected or reallocated. The loop continues iterating using a reference to what is now freed memory — a textbook use-after-free condition.

2.3 Why Was This Especially Dangerous?

Several factors combined to make this a particularly severe bug:

- No JavaScript required: Research analysis suggests the vulnerability can be triggered through CSS @property registration and paint() worklet interactions alone, without any traditional JavaScript payload. This makes it harder for security tools that focus on JS behavior to detect.
- CSS cannot be disabled: Unlike WebGL, WebAssembly, or JIT compilation — all of which can be toggled off — CSS is fundamental to web rendering. There is no browser setting that neuters this attack surface without breaking all web content.
- Low attack complexity: The CVSS vector assigns this a low attack complexity score. Crafting a trigger does not require highly specialized knowledge once the bug class is understood.
- No privileges or user interaction needed beyond visiting a page: An attacker only needs to get the victim to visit a malicious (or compromised) webpage.
- Sandbox escape potential: While the official NVD description confines execution to "inside a sandbox," security researchers note that sandbox escapes are often chained with initial renderer-level RCE. Getting code running inside the renderer process is step one of a multi-stage compromise.

3. The How — Attack Chain & Exploitation

3.1 Attack Flow

A real-world attack using CVE-2026-2441 would proceed in the following stages:

Stage 1 — Delivery	Attacker hosts a crafted webpage (or injects malicious CSS into a legitimate compromised site). Victim is directed to it via phishing, malvertising, watering hole, or redirect.
Stage 2 — Parsing	Chrome's rendering engine (Blink) begins parsing the page's CSS. The malicious @property or font-feature-settings rules trigger the vulnerable ProcessFontFeatures() code path.
Stage 3 — UAF Trigger	The style recalculation cascade frees the FontFeatureMap while iteration is still in progress. The dangling pointer is dereferenced.
Stage 4 — Heap Grooming	The attacker's carefully crafted page layout has pre-positioned attacker-controlled data in the memory location that was freed. The dereferenced pointer now reads or executes attacker data.
Stage 5 — Renderer RCE	Arbitrary code execution is achieved inside the Chrome renderer sandbox. The attacker controls that browser tab's process.
Stage 6 — Sandbox	A separate privilege escalation or sandbox escape exploit (often

Escape (chained)	pre-packaged in a full exploit kit) is used to break out of the sandbox and reach the OS.

3.2 Who Is Being Targeted?

Google has not publicly disclosed details about the threat actors exploiting CVE-2026-2441. However, the broader context gives useful signals:

- The vulnerability was discovered and reported by a single researcher (Shaheen Fazim) on February 11, 2026 — just two days before Google shipped the emergency patch. The speed of the response indicates Google already had intelligence that exploitation was underway.
- The involvement of Google's own Threat Intelligence Group (GTIG) in analyzing related February 2026 vulnerabilities suggests nation-state or highly sophisticated criminal actor involvement.
- The "72-hour double patch" pattern (two major Chrome updates within three days) is consistent with an active exploitation campaign where attackers are pivoting between related bugs.
- Historically, Chrome renderer UAF zero-days have been used in targeted espionage operations and by commercial spyware vendors (e.g., NSO Group-style tooling).

3.3 Why Sandbox Confinement Is Not Enough

A common misconception is that "runs inside the sandbox" means the attack is contained. This is incorrect for several reasons:

- Access to session data: A compromised renderer can exfiltrate everything inside that browser tab — credentials, session cookies, form data, page content, and browser-stored autofill data.
- Lateral movement within the browser: The compromised process can interact with browser APIs, potentially accessing other tabs, extensions, or stored passwords depending on browser isolation mode.
- Sandbox escape primitives exist: Attackers maintaining a library of zero-days routinely chain a renderer RCE with a separate kernel or GPU process escape. The renderer bug is step one, not the final goal.
- JIT spray and info leaks: Even sandboxed code can perform JavaScript JIT spray techniques or memory information leaks that assist in developing a full chain.

4. Remediation — How to Fix This

4.1 Google's Patch

Google's fix is elegant. Instead of iterating directly over the live FontFeatureMap — which can be freed mid-iteration by a cascading style event — the patched code first creates a safe local copy of the map and iterates over that instead:

```
void ProcessFontFeatures(FontFeatureMap& map) {
    // THE FIX: Snapshot the map before iterating
    FontFeatureMap safe_copy = std::move(map);
    for (auto& rule : safe_copy) {  // Iterate the stable copy
        ApplyStyle(rule);            // Even if original is freed, safe_copy
persists
    }
}
```

By moving the data into a local variable (safe_copy), the code ensures that even if the original map is deallocated or modified during ApplyStyle(), the loop is working on a stack-owned snapshot that cannot be freed beneath it. This is a standard and correct UAF mitigation pattern in C++.

4.2 Step-by-Step: Updating Chrome

Windows and macOS

6. Open Google Chrome.
7. Click the three-dot menu (More) in the top-right corner.
8. Navigate to Help > About Google Chrome.
9. Chrome will automatically check for and download available updates.
10. Click Relaunch to apply the update. Do NOT skip this step — the old binary stays active until relaunch even if the download completes.
11. Verify: the version displayed should be 145.0.7632.75 or higher.

Linux

12. Update Chrome via your package manager:

```
# Debian/Ubuntu:
sudo apt update && sudo apt install google-chrome-stable

# Fedora/RHEL:
sudo dnf update google-chrome-stable
```

13. Verify the installed version:

```
google-chrome --version
```

14. Confirm: version should be 144.0.7559.75 or higher on Linux.

Other Chromium-Based Browsers

All browsers built on the Chromium engine share the same vulnerable CSS code path. Update these as well:

Microsoft Edge	Settings > Help and feedback > About Microsoft Edge. Apply and relaunch.
Brave	Menu > Help > About Brave. Update and relaunch.
Opera	Menu > Update & Recovery. Update and relaunch.
Vivaldi	Menu > Help > About. Update and relaunch.
Microsoft Edge (Enterprise)	Deploy via Microsoft Endpoint Manager (Intune) or WSUS patch management.

4.3 Enterprise & Fleet Remediation

For organizations managing many endpoints, manual updates are insufficient. Use the following approach:

Verification Commands

Check the running Chrome version on Windows endpoints (PowerShell):

```
(Get-Item 'C:\Program
Files\Google\Chrome\Application\chrome.exe').VersionInfo.ProductVersion
```

Check on Linux:

```
google-chrome --version
```

Check on macOS:

```
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --version
```

Critical Distinction: Installed vs. Running Version

A patched version being installed does NOT mean the user is protected. Chrome must be relaunched. In an enterprise context, treat these as separate checks:

- Installed version >= 145.0.7632.75 (confirm via MDM/EDR inventory)
- Running process version >= 145.0.7632.75 (confirm via process inspection or EDR telemetry)
- Force a relaunch via group policy or endpoint management tooling if users have not relaunched

Group Policy (Windows Enterprise)

Use Chrome's enterprise ADMX templates to enforce minimum version requirements. Set the ChromeFrameRendererAccessibilityEnabled policy or use a custom policy that blocks Chrome versions below the minimum:

```
// Deploy via GPO or Intune:
HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Google\Chrome
Value: TargetVersionPrefix = "145."
```

4.4 Temporary Workarounds (If Patching Is Delayed)

NOTE	These mitigations are NOT substitutes for patching. They reduce — but do not eliminate — exposure. Apply only if an emergency delay is unavoidable, and patch as soon as possible.

- Block browsing to unknown/untrusted URLs at the network perimeter using DNS or proxy filtering while patching is in progress.
- Enforce Chrome browser management policies that restrict rendering of external CSS from untrusted origins.
- Consider enabling Chrome's site isolation features (--site-per-process) via policy if not already active, which limits the blast radius of a sandboxed exploit.
- Deploy web content inspection / browser isolation platforms if available in your security stack — these render content in a cloud container rather than on the endpoint, preventing memory corruption from reaching local processes.

4.5 Detection: Are You Being Targeted?

Because Google has not released specific IOCs or TTPs for the CVE-2026-2441 exploit campaign, behavioral detection is the primary approach:

- EDR/XDR alerts: Look for Chrome renderer processes (chrome.exe with --renderer flag) spawning child processes, making unusual network calls, or writing to disk outside normal browser cache paths.
- Memory anomalies: Heap spray and UAF exploitation often produce memory access pattern anomalies detectable by modern EDR products with memory protection modules.
- Unusual browser network traffic: Detect POSTs to unexpected external domains from browser processes immediately after page load (possible C2 callback after successful RCE).
- Log browser version enforcement: Identify endpoints still running vulnerable Chrome versions from EDM/MDM inventory — these are your unpatched exposure window.

5. Summary & Key Takeaways

What happened	A use-after-free bug in Chrome's CSS font feature processing allows attackers to execute arbitrary code inside the browser sandbox by getting a victim to visit a crafted webpage.
Why it matters	Actively exploited. No JavaScript required. CSS cannot be disabled. Sandbox containment is insufficient against a determined attacker with a chained exploit.
Who is at risk	All users on Chrome < 145.0.7632.75 on Windows/macOS and < 144.0.7559.75 on Linux. Also Edge, Brave, Opera, Vivaldi, and any Electron app using affected Chromium builds.
How to fix it	Update Chrome immediately and relaunch the browser. Verify both installed and running versions. Update all Chromium-based browsers.
Enterprise priority	Treat as critical. Enforce via MDM/GPO. Verify running versions, not just installed. CISA KEV deadline is March 10, 2026 for federal agencies.
Long-term	Google is integrating Rust into Chromium to eliminate this class of memory safety bug. Until that migration is complete, UAF vulnerabilities in the rendering engine will continue to surface.

Sources: NVD | Google Chrome Releases Blog | CISA KEV Catalog | Malwarebytes | Help Net Security | The Hacker News | SecurityWeek | CrowdStrike | InfoSec Write-ups | Menlo Security